Wintersemester 2016/17

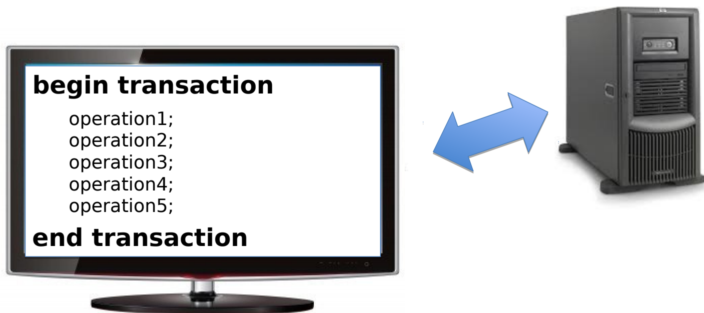
Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Transaktionsverwaltung

Anwendungsprogrammierung: Transaktionen

- Nicht nur eine einzelne SQL-Anweisung, sondern ganze Folge davon, je nach Anwendung.
- Eine oder mehrere Anweisungen werden als Transaktion zusammengefasst bzw. betrachtet. Z.B. Abheben von Geld am Geldautomat.



Literatur

- Grundlagen: **Buch von Kemper und Eickler**: Datenbanksysteme - Eine Einführung. Kapitel 9,10,11 (Ausgabe 8 o.ä.)
- Mehr Details: **Buch von Härder und Rahm**: Datenbanksysteme - Konzepte und Techniken der Implementierung. Kapitel 13–16.

- Aber hier insbesondere: **Buch von Weikum und Vossen**: Transactional Information Systems. Viele Details, Theorie, aber generell sehr gut und anschaulich geschrieben.

Einführung

Bei einer typischen Transaktion in einer Bankanwendung:

1. Lese den Kontostand von A in die Variable a: $read(A,a)$;
2. Reduziere den Kontostand um 50 Euro: $a := a - 50$;
3. Schreibe den neuen Kontostand in die Datenbasis: write(A,a);
4. Lese den Kontostand von B in die Variable b: $read(B,b)$;
5. Erhöhe den Kontostand um 50 Euro: $b := b + 50$;
6. Schreibe den neuen Kontostand in die Datenbasis: $write(B, b)$;

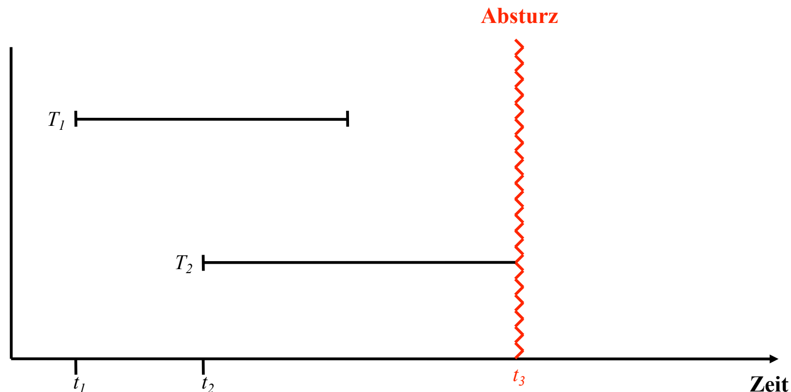
Eigenschaften von Transaktionen: ACID

- **Atomicity (Atomarität):**
  Alles oder nichts
- **Consistency:**
  Konsistenter Zustand der DB $\rightarrow$ konsistenter Zustand
- **Isolation:**
  Jede Transaktion hat die DB “für sich allein”
- **Durability (Dauerhaftigkeit):**
  Änderungen erfolgreicher Transaktionen dürfen nie verloren gehen.,

Operationen auf Transaktions-Ebene

- **begin of transaction (BOT):** Mit diesem Befehl wird der Beginn einer eine Transaktion darstellende Befehlsfolge gekennzeichnet.
- **commit:** Hierdurch wird die Beendigung der Transaktion eingeleitet. Alle Änderungen der Datenbasis werden durch diesen Befehl festgeschrieben, d.h. sie werden dauerhaft in die Datenbank eingebaut.
- **abort:** Dieser Befehl führt zu einem Selbstabbruch der Transaktion. Das Datenbanksystem muss sicherstellen, dass die Datenbasis wieder in den Zustand zurückgesetzt wird, der vor Beginn der Transaktionsausführung existierte.

Transaktionen bei System-Crash



Wie verhält sich solch ein Szenario mit ACID? Was muss beachtet werden?

Abschluss einer Transaktion

Für den **Abschluss** einer Transaktion gibt es **drei Möglichkeiten**:

- Den **erfolgreichen** Abschluss durch **commit**.
- Den **erfolglosen** Abschluss durch ein **abort**.
- Den **erfolglosen** Abschluss durch einen **Fehler**.

Transaktionsverwaltung in SQL

Beispielsequenz auf Basis des Universitätsschemas:

insert into Vorlesungen
values (5275, ‘Kernphysik‘, 3, 2141);
insert into Professoren
values (2141, ‘Meitner‘, ‘C4‘, 205);
commit;

Transaktionsverwaltung in SQL

- **commit [work]:** Die in der Transaktion vollzogenen Änderungen werden – falls keine Konsistenzverletzung oder andere Probleme aufgedeckt werden – festgeschrieben. Das Schlüsselwort work ist optional, d.h. das Transaktionsende kann auch einfach mit commit "befohlen" werden.

- **rollback [work]:** Alle Änderungen sollen zurückgesetzt werden. Anders als der commit-Befehl muss das DBMS die "erfolgreiche" Ausführung eines rollback-Befehls immer garantieren können.

Sicherungspunkte

- **Sicherungspunkt:**
  Punkt innerhalb einer TA, auf den sich aktive TA zurücksetzen lässt
- **savepoint <name>:**
  definiert den Sicherungspunkt
- **rollback [work] to <name>:** setzt aktive TA zurück bis zum Sicherungspunkt <name>

Beispiel

```
begin;
insert into tab values ...
savepoint A;
insert into tab values ...
savepoint B;
SELECT * FROM tab;
rollback to A;
SELECT * FROM tab;
...
```

Wie unterstützt das DMBS Transaktionen?

Mehrbenutzersynchronisation (Isolation)

- (Ausführlich in der VL Informationssysteme behandelt)
- semantische Korrektheit bei Nebenläufigkeit
- Serialisierbarkeit
- Schwächere Isolationsstufen (Isolation Levels)

Recovery (Atomicity and Durability)

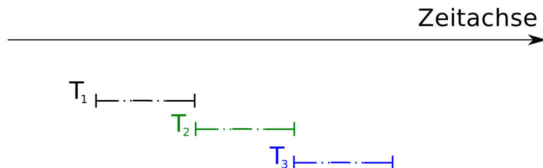
- Zurücksetzen teilweise ausgeführter TA
- Vervollständigen der Aktionen von TA
- Sicherstellen der Persistenz von TA

Wiederholung: Mehrbenutzersynchronisation

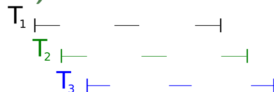
Das “**I**” in AC**I**D.

Ausführung der drei Transaktionen T_1, T_2 und T_3:

(a) im Einzelbetrieb



(b) im (verzahnten) Mehrbenutzerbetrieb



Ziel: Semantik von seriell ausgeführten Transaktionen zusammen mit Performance von verzahnter Ausführung.

Wiederholung: Das lost-update Problem

t_1	Time	t_2
	/* x = 100 */	
r(x)	1	
	2	$r(x)$
/*update $x := x + 30$ */	3	
	4	/* update $x := x + 20$ */
$w(x)$	5	
	/* $x = 130$ */	
	6	$w(x)$
	/* $x = 120$*/	

Wiederholung: Das **lost-update Problem** / Notation

- Die Essenz dieses Problems kann durch folgende Sequenz von Lese- und Schreiboperationen ausgedrückt werden:

$$r_1(x)r_2(x)w_1(x)w_2(x)$$

- $r_i(x)$ **beschreibt das Lesen von Datensatz x durch Transaktion i**
- $w_i(x)$ **analog das Schreiben von Datensatz x durch Transaktion i**

Konflikte zwischen Transaktionen können auch auftreten, wenn eine der beiden TA nur liest - wie im folgenden Beispiel klar wird.

Wiederholung: Das **inconsistent-read Problem**

Beispiel aus z.B. Anwendung in Bank. Aktueller Stand $x = y = 50$, also $x + y = 100$. Transaktion t_1 berechnet die Summe von x und y, während t_2 einen Wert von 10 von x nach y transferiert.

t_1	Time	t_2
	1	$r(x)$
	2	/* $x := x - 10$ */
	3	$w(x)$
/* $sum := 0$ */	4	
$r(x)$	5	
$r(y)$	6	
/* $sum := sum + x$ */	7	
/* $sum := sum + y$ */	8	
	9	$r(y)$
	10	/* $y := y + 10$ */
	11	$w(y)$

Wiederholung: Das **inconsistent-read** Problem (2)

- Offensichtlich ist auch hier wieder das Problem, dass Lese- und Schreiboperationen der einzelnen Transaktionen gemischt ablaufen

$$r_2(x)w_2(x)r_1(x)r_1(y)r_2(y)w_2(y)$$

Wiederholung: Das **dirty-read** Problem

t_1	Time	t_2
$r(x)$	1	
/* $x := x + 100$ */	2	
$w(x)$	3	
	4	$r(x)$
	5	/* $x := x - 100$ */
failure & rollback	6	
	7	$w(x)$

In Vorlesung InSy bereits betrachtet

Transaktionen Konzept

- Atomare Lese- und Schreiboperationen (auf Datenobjekte)
- Transaktion als endliche Folge von Operationen p_i
  $T = p_1 p_2 p_3 \ldots p_n$ mit $p_i \in \{r(x_i, w(x_i)\}$
- TA hat als letzte Operation entweder Abbruch a oder Commit c

Serialisierbarkeit

- Betrachtung von Historien über verschiedenen Transaktionen
- Ziel: Gibt es eine äquivalente serielle Historie?

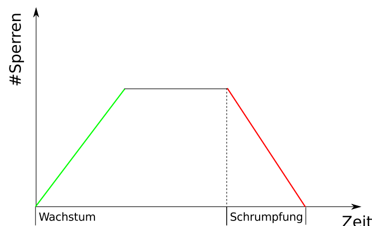
In Vorlesung InSy bereits betrachtet (2)

Sperrbasierte Synchronisation

- shared Lock (Lesesperre)
- exclusive Lock (Schreibsperre)
- Verträglichkeitsmatrix (auch Kompatibilitätsmatrix genannt)

Zwei-Phasen Sperrprotokoll (2PL)

- 2PL erzeugt nur serialisierbare Historien:



- Weitere Protokolle wie striktes 2PL

JDBC - Transaktionen

Transaktionen

- Bei der Erzeugung eines Connection-Objekts ist (in der Regel) als Default der Modus **autocommit** eingestellt. D.h. nach jeder Aktion wird ein Commit ausgeführt.
- Um Transaktionen als Folgen von Anweisungen abwickeln zu können, ist dieser Modus auszuschalten.

```
conn.setAutoCommit(false);
```

- Für eine Transaktion können sogenannte Konsistenzstufen (isolation levels) wie TRANSACTION_SERIALIZEABLE, TRANSACTION_REPEATABLE_READ usw. eingestellt werden.

```
conn.setTransactionIsolation(
        Connection.TRANSACTION_SERIALIZABLE
 );
```

JDBC - Transaktionen (2)

Beendigung oder Zurücksetzung

```
conn.commit();
```

bzw.

```
conn.rollback();    //oder: conn.rollback(savepoint)
```

Sicherungspunkte (Savepoints)

```
Savepoint sp = conn.setSavepoint();  //bzw. mit Namen
Savepoint namedSp = conn.setSavepoint("mySavePoint");
```

Programm kann mit mehreren DBMS verbunden sein

- Selektives Beenden/Zurücksetzen von Transaktionen pro DBMS
- Kein global atomares Commit möglich

JDBC - Transaktionen: Beispiel

http://www.tutorialspoint.com/jdbc/jdbc-transactions.htm

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();
   String SQL = "INSERT INTO Employees  " +
        "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
              "VALUES (107, 22,  'Sita', 'Singh')";

   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

Zusammenfassung Concurrency ;)

https://www.youtube.com/watch?v=G3xH2SoMOF0

Recovery

Durability? – Beispiel 1a

- TA ändert Daten im Hauptspeicher
- Daten noch **gar nicht** auf Festplatte geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- ⇒ Annahme des Benutzers: Änderungen der TA sind dauerhaft in DB gespeichert
- Stromausfall
- Was passiert?
- Welche Daten befinden sich in der DB?

Durability? – Beispiel 1b

- TA ändert Daten im Hauptspeicher
- Daten **teilweise** auf Festplatte geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- $\Rightarrow$ Annahme des Benutzers: Änderungen der TA sind dauerhaft in DB gespeichert
- Stromausfall
- Was passiert?
- Welche Daten befinden sich in der DB?

Durability? – Beispiel 2a

- TA ändert Daten im Hauptspeicher
- Daten **vollständig** auf Festplatte geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- $\Rightarrow$ Annahme des Benutzers: Änderungen der TA sind dauerhaft in DB gespeichert
- Hardwarefehler $\Rightarrow$ Totalverlust der Festplatte
- Was passiert?
- Welche Daten befinden sich in der DB?

Durability? – Beispiel 2b

- TA ändert Daten im Hauptspeicher
- Daten **vollständig** auf **mehrere** Festplatten geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- ⇒ Annahme des Benutzers: alle Änderungen der TA sind dauerhaft in DB gespeichert
- Wasserschaden/Feuer/Erdbeben ...
- ⇒ Totalverlust **aller** Festplatten
- Was passiert?
- Welche Daten befinden sich in der DB?

Durability? – Beispiel 2c

- TA ändert Daten im Hauptspeicher
- Daten **vollständig** auf **mehrere** Festplatten and **mehreren geographisch verteilten** Rechenzentren geschrieben
- Transaktion schickt commit
- Benutzer glaubt, dass TA abgeschlossen ist
- $\Rightarrow$ Annahme des Benutzers: alle Änderungen der TA sind dauerhaft in DB gespeichert
- Wasserschaden/Feuer/Erdbeben ... an allen Rechenzentren gleichzeitig
- $\Rightarrow$ Totalverlust **aller** Rechenzentren **und** Festplatten
- Was passiert?
- Welche Daten befinden sich in der DB?

Quintessenz

- Durability immer relativ
- bezogen auf Anzahl Kopien/geographische Verteilung
- Garantie bezogen auf relative Durability kann nur gemacht werden, wenn
  - erst die verschiedenen Kopien erzeugt werden und
  - dann dem Benutzer mitgeteilt wird, dass die TA committed wurde

$\Rightarrow$ **WAL-Prinzip (write-ahead logging)**

- Varianten hiervon:
  - **log-basierte Recovery**
  - komplettes Spiegeln (mehrere Rechner/Rechenzentren machen dasselbe redundant)

Welche Aspekte von

ACID

werden durch Recovery adressiert?

Implikationen von ACID auf Anforderungen zu Recovery

Durability

- Änderungen an der Datenbank, die durch erfolgreich(!) abgeschlossene (d.h. committed) Transaktionen verursacht wurden, müssen dauerhaft gespeichert sein.
- D.h. bei einem Crash der DB muss nach Wiederanlauf geschaut werden, ob dies tatsächlich der Fall ist.

Atomicity

- Falls eine Transaktion noch nicht erfolgreich abgeschlossen wurde und ein DB-Crash auftritt, muss beim Wiederanlauf darauf geachtet werden, dass etwaige Änderungen in der Datenbasis rückgängig gemacht werden.

Fehlerklassifikation

Lokaler Fehler in einer noch nicht festgeschriebenen (committed) Transaktion:

- Wirkung der TA muss zurückgesetzt werden

Fehler mit Hauptspeicherverlust:

- Abgeschlossene TAs müssen erhalten bleiben
- Noch nicht abgeschlossene TAs müssen zurückgesetzt werden

Fehler mit Hintergrundspeicherverlust:

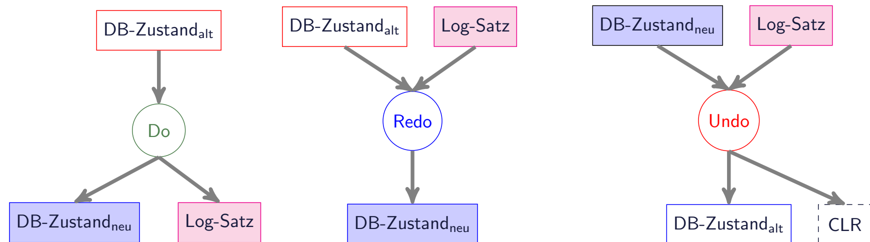
- Archiv einspielen

Vorsorge für den Fehlerfall

Logging

- Sammlung redundanter Daten bei Änderungen im Normalbetrieb, als Voraussetzung für Recovery
- Einsatz im Fehlerfall (Undo-, Redo-Recovery)

Do-Redo-Undo-Prinzip



Recovery-Oriented Computing

Systemverfügbarkeit A (availability)

- MTTF: Mean Time To Failure
- MTTR: Mean Time To Repair

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Warum Recovery-Oriented Computing?

- Hardwarefehler, Softwarefehler passieren (sind nicht vermeidbar) und müssen adressiert werden.
- Lange Systemausfälle sind sehr sichtbar (z.B. Amazon, Facebook, Ebay!)

Number of Nines & Entwicklungsziele

Availability wird manchmal beschrieben in Anzahl von Neunen (Nines), wie in “five nines”, oder 99.999%. 99.99% entspricht ungefähr 1 Minute Ausfall in einer Woche, bzw. circa 50 Minuten Ausfall in einem Jahr.

- “Build a system used by millions of people that is always available – out less than 1 second per 100 years = 8 9’s of availability” (J. Gray: 1998 Turing Award Lecture)

Jim Gray: We have added three 9s in 45 years (starting with 90%), or about 15 years per order-of-magnitude improvement in availability. We should aim for five more 9s: an expectation of one second outage in a century. This is an extreme goal, but it seems achievable if hardware is very cheap and bandwidth is very high. One can replicate the services in many places, use transactions to manage the data consistency, use design diversity to avoid common mode failures, and quickly repair nodes when they fail. Again, this is not something you will be able to test: so achieving this goal will require careful analysis and proof.

Quelle: bild.de, 27.01.2015

- Ausfall hat (angeblich) ca. 1 Stunde gedauert
- Nehmen wir an, dass dies ein Mal im Jahr geschieht.
- Wie groß ist die Availability? Wie viele "Neunen"?

Wie kann man annährend $A = 1{,}0$ erreichen?

- MTTF $\rightarrow \infty$?
- MTTR $<<$ MTTF!

Es gibt Fehler die man nur sehr schwer oder gar nicht nicht durch testen in den Griff bekommen kann. Sie treten nichtdeterministisch auf, oft aufgrund von Nebenläufigkeit in Threads, unter hoher Last, oder in anderen seltenen Fällen.

Solche Probleme werden auch "Heisenbugs" genannt (Jim Gray); nach Heisenbergs Unschärferelation.

D.h. ein schneller Wiederanlauf (Recovery) ist essentiell für hohe Verfügbarkeit (Availability); da diese "Heisenbugs" die MTTF in der Praxis einschränken.

Grundlagen der DB-Recovery

Aufgabe des DBMS

- Automatische Behandlung aller erwarteten Fehler

Was sind erwartete Fehler?

- DB-Operation wird zurückgewiesen, Commit wird nicht akzeptiert, ...
- Stromausfall, DBMS-Probleme
- Geräte funktionieren nicht (Spur, Zylinder, Platte defekt)
- Beliebiges Fehlverhalten der Gerätesteuerung
- ...

Fehlermodelle von (zentralisierten) DBMS

- Transaktionsfehler
- Systemfehler
- Gerätefehler
- Katastrophen

Recovery-Arten

Transaktions-Recovery

- Zurücksetzen einzelner (noch nicht abgeschlossener) TA im laufenden Betrieb (TA-Fehler, Deadlock, etc.)

→ Vollständiges Zurücksetzen auf BOT (TA-Undo)

→ Partielles Zurücksetzen auf Rücksetzpunkt (Savepoint) innerhalb der Transaktion

Crash-Recovery nach Systemfehler

- Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustands:

→ (partielles) Redo für erfolgreiche TA (Wiederholung verlorengegangener Änderungen

→ Undo aller durch Ausfall unterbrochenen TA (Entfernung der Änderungen aus der DB)

Recovery-Arten (2)

Medien-Recovery nach Gerätefehler

- Spiegelplatten, bzw.
- vollständiges Wiederholen (Redo) aller Änderungen auf einer Archivkopie

Katastrophen-Recovery

- Nutzung einer aktuellen DB-Kopie in einem “entfernten” System oder
- stark verzögerte Fortsetzung der DB-Verarbeitung mit repariertem/neuem System auf Basis gesicherter Archivkopien

Fehlermodell in kommenden Vorlesungen

- Wir betrachten im Folgenden den Fall eines System-Crashs mit Verlust des Hauptspeichers.
- D.h. Festplatte (Hintergrundspeicher) ist stabil und verliert keine Daten.

Danach wird auch Transaktions-Recovery betrachtet.

Crash-Recovery

Im folgenden wird Crash-Recovery betrachtet.

Idee/Ziel

- **Wiederanlauf (Restart) des DBS nach einem Systemfehler**
- Ziel: Nach Wiederanlauf den jüngsten transaktionskonsistenzen DB-Zustand wiederherstellen, der zum Fehlerzeitpunkt gültig war.
- **Dazu wird materialisierte Datenbasis und Log-Datei benutzt.**
- Zusätzliche Anforderung: **Idempotenz!** D.h. bei mehrfacher Anwendung der Recovery muss dies stets zum selben Ergebnis führen.

Abhängigkeit von System-Konfiguration

- Methoden zur Crash-Recovery sind wesentlich durch die zugrunde liegende System-Konfiguration bestimmt (Satz oder Seitensperren, steal oder no steal, etc), wie auf folgenden Folien motiviert.

Undo und Redo

Pufferinhalt geht verloren, was dann?

Undo

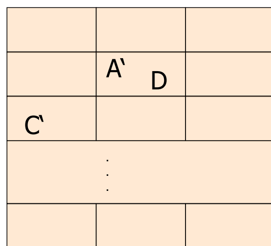
Alle durch nicht abgeschlossene Transaktionen schon in die materialisierte Datenbasis eingebrachten Änderungen müssen rückgängig gemacht werden.

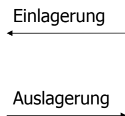
Redo

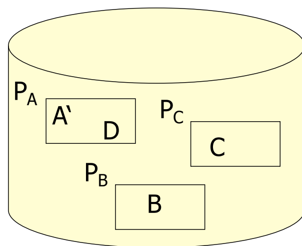
Alle noch nicht in die materialisierte Datenbasis eingebrachten Änderungen durch abgeschlossene Transaktionen müssen nachvollzogen werden.

Zweistufige Speicherhierarchie

- Seiten P_i
- Datensätze A,B,C,D, ..
- **Werden später den Fall betrachten, dass Datensätze gesperrt werden, also versch. TA die gleiche Seite bearbeiten können.**







Einbringungsstrategie

Update in Place:

- jede Seite hat genau eine “Heimat” auf dem Hintergrundspeicher
- der alte Zustand einer Seite wird überschrieben

Twin-Block-Verfahren:

- Jede Seite hat zwei Versionen
- Anordnung für Seiten P_A, P_B und P_C

P_A^0	P_A^1	P_B^0	P_B^1	P_C^0	P_C^1	...

Schattenspeicherkonzept:

- nur geänderte Seiten werden dupliziert
- weniger Redundanz als beim Twin-Block-Verfahren

Die Speicherhierarchie

Ersetzung von Puffer-Seiten:

- **¬steal:** Ersetzung von Seiten, die von einer noch aktiven Transaktion modifiziert wurden, ausgeschlossen ⇒ “dreckige” Seiten (dirty pages) müssen im Puffer bleiben.
- **steal:** Jede nicht fixierte Seite ist prinzipiell ein Kandidat für die Ersetzung, falls neue Seiten eingelagert werden müssen, auch “dreckige” Seiten.

Einbringen von Änderungen abgeschlossener TAs:

- **force:** Änderungen werden bei commit auf den Hintergrundspeicher geschrieben (flush).
- **¬force:** geänderte Seiten können auch nach commit im Puffer verbleiben.

dirty page = Inhalt der Seite im HS ≠ Inhalt der Seite auf FS

Auswirkungen auf Recovery

Undo: entfernt ungültige Einträge aus der DB.

Redo: fügt gültige Einträge in die DB sein.

	force	**¬force**
¬steal		
steal		

- Was ist besser? steal oder ¬steal?
- Was ist besser? force oder ¬force?
- Annahme: Schreiboperationen von Seiten müssen atomar sein.

Auswirkungen auf Recovery

Undo: entfernt ungültige Einträge aus der DB.

Redo: fügt gültige Einträge in die DB sein.

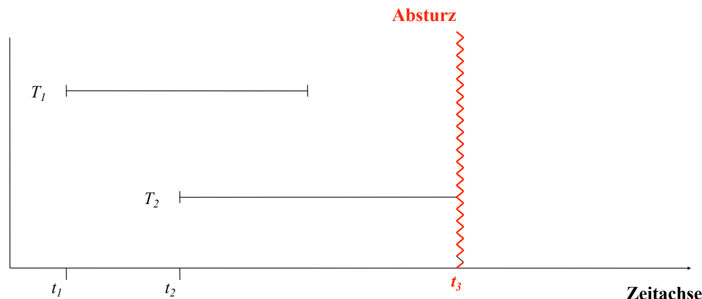
	force	**¬force**
¬steal	• kein Undo • kein Redo ⇒ **keine** Recovery	• Redo • kein Undo
steal	• kein Redo • Undo	• Redo • Undo

- Was ist besser? steal oder ¬steal?
- Was ist besser? force oder ¬force?
- Annahme: Schreiboperationen von Seiten müssen atomar sein.

Hier zugrunde gelegte Systemkonfiguration

- **steal**
  "dreckige Seiten" können in die Datenbank (auf Platte) geschrieben werden.
- **¬force**
  geänderte Seiten sind **möglicherweise** noch nicht auf die Platte geschrieben
- **update-in-place**
  Es gibt von jeder Seite nur eine Kopie auf der Platte
- **Kleine Sperrgranulate, kleiner als eine Seite**
  auf Satzebene. Also kann eine Seite gleichzeitig "dreckige" Daten (einer noch nicht abgeschlossenen TA) und "committed updates" enthalten.

Wiederanlauf nach einem Fehler



- Transaktionen der Art T_1 müssen hinsichtlich ihrer Wirkung vollständig nachvollzogen werden. Diese TAs nennt man **Winner**.
- Transaktionen, die wie T_2 zum Zeitpunkt des Absturzes noch aktiv waren, müssen rückgängig gemacht werden. Diese TAs nennt man **Loser**.

Beispiel einer Log-Datei

Schritt	T_1	T_2	Log-Record [LSN, TA, PageID, Redo, Undo, PrevLSN]
1.	**BOT**		[#1,T_1, **BOT**, 0]
2.	r(A,a1)		
3.		**BOT**	[#2,T_2, **BOT**, 0]
4.		r(C,c2)	
5.	a1 := a1 - 50		
6.	w(A,a1)		[#3,T_1,PA,A-=50,A+=50,#1]
7.		c2 := c2 + 100	
8.		w(C,c2)	[#4,T_2,PC,C+=100,C-=100,#2]
9.	r(B,b1)		
10.	b1 := b1 + 50		
11.	w(B,b1)		[#5,T_1,PB,B+=50,B-=50,#3]
12.	**commit**		[#6,T_1,**commit**,#5]
13.		r(A,a2)	
14.		a2 := a2 - 100	
15.		w(A,a2)	[#7,T_2,PA,A-=100,A+=100,#4]
16.		**commit**	[#8,T_2,**commit**,#7]

WAL-Prinzip und Commit-Regel bei Log-basierter Recovery

Write-Ahead-Log-Prinzip (WAL)

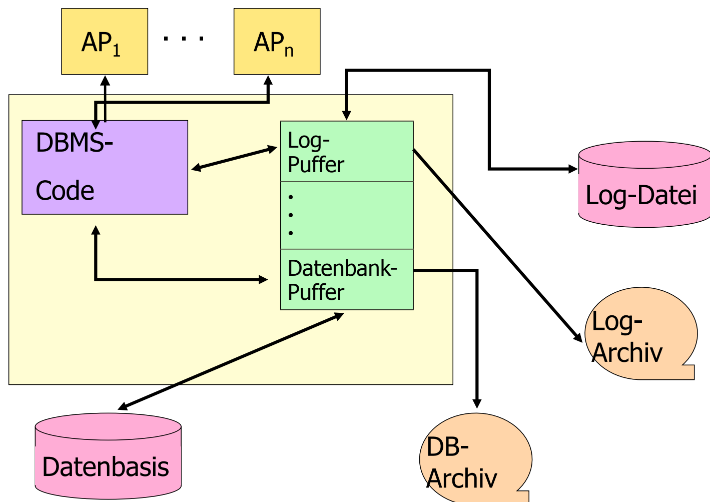
Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in die Log-Datei und das Log-Archiv ausgeschrieben werden.

Commit-Regel (Force-Log-at-Commit)

Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle “zu ihr gehörenden” Log-Einträge auf stabilen Storage ausgeschrieben werden.

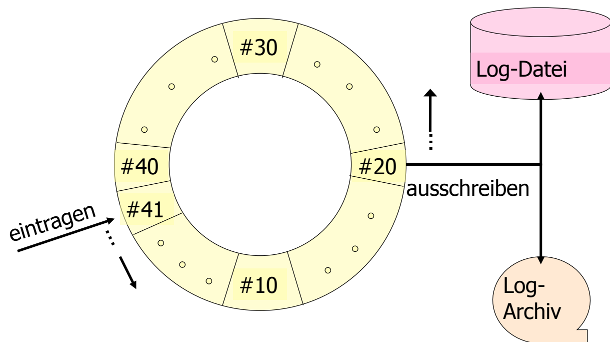
C. Mohan et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.
`http://www.cs.berkeley.edu/~brewer/cs262/Aries.pdf`

Schreiben von Log-Informationen



- Log-Informationen werden zweimal geschrieben: Log-Datei für schnellen Zugriff und Log-Archiv

Anordnung des Log-Ringpuffers



- Kontinuierliches Ausschreiben
- Aber Achtung: WAL und Commit-Regel beachten!
- Log-Puffer ist normalerweise kleiner als DB-Puffer
- Groß genug um i.d.R. laufende Transaktionen zu enthalten, so dass beim Rücksetzen einer TA dies anhand des Puffers gemacht werden kann.

Klassifikation von Logging-Verfahren

Wie können Redo- und Undo-Informationen protokolliert werden?